# Homework Assignment 3
# Live public transport monitoring/tracking system – Simulated movement, data management & OOP with advanced features

CSAS 2124 - Spring 2025
Due date: 4/22/2025
Submission: .zip of all code files + README + test results + screenshots

In this assignment, you will build an extensible, modular, and real-time public transport monitoring backend system to simulate how a smart city manages its transit operations. Your backend system must simulate four types of vehicles - each modeled as a networked client - communicating with a central control server using TCP and UDP sockets. All entities operate over a local network, emulating real-world vehicle telemetry, command-and-control behavior, and live location tracking.

Imagine it's Monday, 7:30am in New York City. The city's Department of Urban Mobility is piloting a state-wide Intelligent Public Transport Monitoring System (IPTMS) to monitor its vehicles in real-time. Your job is to design a backend simulation system where each vehicle type is represented by a networked client, and there is one central server running at the control center near Bryant Park. All entities communicate across a distributed network, mimicking real-world public transport telemetry, command-and-control protocols, and movement behavior.

**Vehicles in action:**

**Bus B101 - Midtown to financial district loop (Route R1)**

- Leaves Port Authority Terminal and heads down via 7th Ave, stopping at Times Square, Flatiron, and Wall Street.
- Every 1 minute, it updates its location and status via TCP to the central server.
- If it experiences simulated congestion near Union Square, the control center sends a DELAY command, which must be acknowledged and applied.
- The bus moves sequentially, stop-by-stop, and resumes only when told to.
- Sends a UDP beacon every 10 seconds to broadcast its latest coordinates for the public dashboard (stateless).

NYC buses use systems like MTA BusTime which ping control centers and apps with status and location updates

What the passenger sees:

**Bus B101 | En route to Wall Street | Current stop: Flatiron | Status: On Time | ETA to next stop: 2 mins**

**Train T22 - MTA M Line (Queens – Midtown - Brooklyn)**

- Leaves Queens Plaza, enters Herald Square, and stops at Delancey St before heading to Middle Village.
- Only reports on arrival at major stations, simulating bandwidth optimization in rail telemetry systems.
- It uses TCP for critical event updates (arrival, delay, reroute).
- If the control center sends a SHUTDOWN command due to an incident ahead, the train client must safely stop and enter standby mode.
- Observers (e.g., the city log system) must be notified of every major event-this reflects an Observer pattern usage.

Systems like NYCT Subway's ATS (Automatic Train Supervision) only report train location at key intervals and log critical events. What the passenger sees:

**Train T22 | Departed: Queens Plaza | Next stop: Herald Square | Status: Delayed | ETA: 4 mins**

**Uber U991 - Washington square to Columbia University**

- It starts from a random spot near NYU, with the destination set near Columbia's campus.
- Generates a dynamic path using your simulation logic, with no fixed stops.
- Sends its position every 5 seconds over UDP, without expecting acknowledgment.
- The admin cannot interrupt or reroute it mid-journey, respecting encapsulated business rules.
- During simulation, simulate a network dropout near the Lincoln Tunnel. Your system should pause updates and retry silently until the TCP connection is re-established.

Similar to Uber's real-time GPS update logic and passenger privacy model where internal systems log rides but do not expose user-specific routing externally. What the passenger sees:

**Uber U991 | In Transit | Approximate location: Near Bryant Park | Status: Active**

**Airport shuttle S01 – Penn station to JFK (Every 30 minutes)**

- Runs only at scheduled intervals. Until 8:00 AM, it must stay in "Standby" mode.

- At 8:00 AM, it receives a START_ROUTE command over TCP, transitions into “Active,” and begins updates.
- The admin mistakenly sends the command early at 7:45 AM. The Shuttle must ignore it.
- Location updates occur over UDP, status changes and command acknowledgments occur via TCP.
- Demonstrates the Command pattern, where the server issues requests that they must be validated and acted upon only if permissible.
- Simulates JFK AirTrain or GO Airlink Shuttle systems, which only start tracking after activation by a scheduler or dispatcher.

What the passenger sees:

**Shuttle S01 | Status: Standby (Next Departure: 8:00 am)**

**System design implications**

Your simulation must reflect the modular complexity of real-world transport systems. Though it runs on local networks, it must simulate concurrency, reliability, and fault resilience as found in distributed fleet.

**Independent Clients (Multithreaded/Multi-process architecture)**

Each vehicle in your simulation - Bus, Train, Uber, or Airport Shuttle - must be implemented as a self-contained client, running as a separate process or thread. This models the autonomy of transport vehicles in a real urban setting.

Your implementation must ensure the following:

- Clients operate independently: No vehicle depends on another for movement or decisions. Each client should simulate its path, report its own status, and respond to commands from the server on its own timeline.
- Asynchronous communication: Use threading, multiprocessing, or asyncio to send and receive data independently across all clients and the server. Communication should not block any other component.
- Fault isolation: A crash, delay, or disconnection in one vehicle client must not affect the operation of others. Clients must operate in isolation with proper error handling.

In real-world systems, a train delay should not stop a bus or an Uber from operating. Your simulation must reflect this independence by design.

**TCP Sockets for reliable command & control**

You must use TCP sockets for any reliable, bidirectional communication between the central server and vehicle clients. TCP is required wherever guaranteed message delivery, ordering, or acknowledgment is needed.

TCP must be used for:

- Status updates: Vehicles should send structured messages (vehicle ID, location, and status like “On Time”, “Delayed”) at defined intervals to the server.
- Receiving admin commands: The server issues critical commands (e.g., DELAY, REROUTE, SHUTDOWN), which clients must receive, validate, acknowledge, and execute if permitted.
- Reconnection logic: Clients that lose TCP connection must detect the failure, reconnect to the server, and resume reporting and command listening.
- Eg., You can simulates backend communication in systems like **MTA's Control Center**, where critical messages are never dropped.

**Why do we use TCP?**

- Guarantees ordered delivery.
- Ensures no message loss during transmission.
- Allows acknowledgment and synchronized state between client and server, especially critical for commands affecting public safety or simulation logic.

**UDP Sockets for lightweight, high-frequency updates**

Your system must also use UDP sockets for periodic, stateless location updates messages that simulate what real-time public dashboards or lightweight trackers receive.

UDP should be used for:

- **Location update**: Every vehicle must broadcast its latest coordinates at a fixed short interval (e.g., every 5 - 10 seconds).
- **Dashboard updates**: These messages can be visualized or logged for simulation purposes. They do not require acknowledgment or response from the server.
- **Passive status**: Vehicles in standby mode (e.g., the airport shuttle before 8:00am) can use UDP to periodically say, I’m alive, but not active.
- Eg., similar to how GPS trackers or bus signage systems relay data frequently without needing acknowledgments.

**Why UDP?**

- Faster, low-overhead, and scalable for frequent updates.
- No need for connection establishment (no TCP handshakes).

- Common in GPS systems and vehicle telemetry where approximate updates are acceptable and need to scale with many clients.

Your system must implement design patterns that solve real architectural challenges in the simulation. Patterns are not for show - they must add meaningful functionality. If your system behaves incorrectly, inefficiently, or is harder to extend without the pattern - then you've chosen and applied the pattern well.

**Required design patterns and their functional use**

- Choose the right design pattern from Creational and structural design pattern.
- For Behavioral pattern:
    - Observer - Implement a mechanism where the dashboard, log system, or admin monitor subscribes to updates from vehicles. Whenever a vehicle reaches a major event (arrival, shutdown), notify all observers. This ensures real-time updates without tight coupling.
    - Command - Admin commands like DELAY, SHUTDOWN, or START_ROUTE should be encapsulated as command objects, sent from the server, and executed by the vehicles. This pattern decouples the logic of "what to do" from "how to do it," and allows command queuing, rejection, or rollback.

**Server simulated control actions (Real-time command engine)**

The central controller (server) is the brain of the IPTMS and must simulate an admin-level control interface. This component must be able to:

**Behavior & functionality**

- Accept TCP connections from each incoming client.
- Maintain a registry (dictionary/map) of connected vehicle IDs and their TCP sockets.
- Receive and interpret admin commands (via CLI or file).
- Package commands (using Command pattern), and dispatch them to the correct client.
- Log:
    - Which commands were issued
    - Whether they were acknowledged, failed, or rejected
    - Current status of all active vehicles

**Supported Admin Commands (Examples)**

- DELAY T22 20 -> Delays Train T22 for 20 seconds. Train pauses and resumes after duration.

- REROUTE B101 -> Requests Bus B101 to deviate from the standard route. Bus switches its strategy.
- SHUTDOWN U991 -> Instructs Uber U991 to exit simulation and notify dashboard.
- START_ROUTE S01 -> Activates Shuttle S01 at or after its scheduled time. Before that, the command is rejected.

All commands must be:

- Encapsulated (Command pattern)
- Logged in detail
- Responded to by the client vehicle
- Executed only if permitted (based on time, type, etc.)

**Resilience and Recovery (Fault simulation)**

Your simulation must gracefully handle real-world failure scenarios. These tests will prove your system's robustness, separation of concerns, and recovery architecture.

**Faults you must simulate & handle**

**1. Network outages**

- A vehicle may lose its TCP connection temporarily.
- It must retry automatically without blocking or crashing.
- Upon reconnect, it should resume normal updates and command listening.

Implement retry logic using exponential backoff, and maintain client state for reconnect.

**Late join (Vehicle startup mid-simulation)**

- Some clients (e.g., Shuttle S01) may start after the server is already live.
- Server should accept new TCP connections and integrate them into the registry.
- Late vehicles should start broadcasting real-time location updates (including vehicle ID, latitude, longitude, and timestamp) over UDP once they become active. Use a handshake or welcome message to sync state on connect.

**3. Missed Commands**

If a command is sent while a client is offline or unresponsive:

- The server must timeout the delivery
- Optionally queue the command for retry

- Or log the failure and notify the admin

Use a message wrapper with delivery status fields like "delivered", "acknowledged", or "expired".

**4. Out-of-policy actions**

The system must reject invalid commands:

- Example: Shuttle S01 receives START_ROUTE at 7:45 AM (before its 8:00 AM scheduled time)
- The command is logged as rejected, and the shuttle remains in standby

This shows proper use of business rule enforcement and command pattern filtering.

**Final system behavior**

Your system must:

- Stay running throughout faults
- Log and reflect all vehicle activity
- Maintain communication integrity (no command duplication or data corruption)
- Cleanly handle disconnects, reconnects, and command retries
- Visibly separate internal control logic from public-facing (passenger) updates

**Data management with SQLite3**

To support real-time monitoring, historical tracking, and system resilience, your assignment must include a thread-safe, persistent database layer using SQLite3. This database will simulate how real-world transport systems store and retrieve telemetry data, command history, vehicle behavior, and system faults. Your system must log data continuously from both TCP and UDP channels, and the control center should be able to query this data for analysis and CLI-based dashboard displays.

**Required tables (minimum – add/modify based on your idea):**

- **vehicles**: Stores metadata and current state for each vehicle.
  Fields: vehicle_id, vehicle_type, route_id, status, last_seen, etc.
- **routes**: Stores route details (static)
  Fields: route_id, origin, destination, stop_sequence
- **location_updates**: Logs real-time data from each vehicle
  Fields: update_id, vehicle_id, latitude, longitude, speed, timestamp, network_status

- **admin_commands**: Tracks all admin command activity
  Fields: command_id, vehicle_id, command_type, parameters, sent_time, response_time, status
- **event_logs**: Logs system-level or fault events
  Fields: event_id, vehicle_id, event_type, details, event_time

**Implementation requirements:**

- Use *parameterized SQL statements* for inserts and updates.
- All database interactions must be concurrency-safe using appropriate locking (threading.Lock()).
- The database must log:
  - All TCP status updates and acknowledgments
  - All UDP location beacons
  - All commands issued and responses
  - All events, including network outages, missed commands, or invalid actions
- Implement archival functionality: location updates older than 15 minutes must be written to a .csv file during or at the end of simulation.

**Required query capabilities:**

You must demonstrate and document at least three meaningful SQL queries, such as:

1. Vehicles currently in "Delayed" state
2. List of all SHUTDOWN commands issued and their outcomes
3. Average response time to admin commands, grouped by vehicle

**Submission guidelines: Required files:**

- Source code files (.py, or relevant project directory)
- README.md including:
  - Setup instructions
  - Design decisions
  - Explanation of all implemented design patterns
- Sample output files (TCP, UDP logs, and screenshots)
- At least 3 test cases with input/output

- Optional: system architecture diagram or flowchart
- Submit everything as a single .zip file named: CSAS2124_HW2_LastnameFirstname.zip

**Sample output**

[07:30:00] SERVER STARTED at Bryant Park Control Center
[07:30:05] Vehicle CONNECTED: B101 (Bus) via TCP
[07:30:07] Vehicle CONNECTED: T22 (Train) via TCP
[07:30:10] Vehicle CONNECTED: U991 (Uber) via TCP

[07:30:15] [TCP] B101 -> Status Update:
    Location: (40.7510, -73.9900) | Status: On Time

[07:30:16] [UDP] B101 -> Real-Time Location Update:
    Lat: 40.7510 | Long: -73.9900 | Status: On Time

[07:30:20] [TCP] T22 -> Status Update:
    Location: (40.7482, -73.9850) | Status: On Time

[07:30:22] [UDP] U991 -> Real-Time Location Update:
    Lat: 40.7290 | Long: -73.9982 | Status: Active

[07:30:30] [COMMAND] DELAY issued to T22 for 20 seconds
[07:30:31] [ACK] T22 -> Acknowledged DELAY. Status updated to: Delayed

[07:30:35] [UDP] B101 -> Real-Time Location Update:
    Lat: 40.7520 | Long: -73.9875 | ETA to next stop: 2 mins

[07:30:38] [UDP] U991 -> Real-Time Location Update:
    Lat: 40.7315 | Long: -73.9970 | ETA: 6 mins

[07:30:40] [TCP] B101 -> Status Update:
    Location: (40.7520, -73.9875) | Status: On Time

[07:30:41] [COMMAND] SHUTDOWN issued to U991
[07:30:42] [REJECTED] U991 -> SHUTDOWN not permitted (private ride - encapsulated rules)

[07:30:55] Vehicle CONNECTED: S01 (Shuttle) via TCP
[07:30:57] [UDP] S01 -> Broadcasted passive status:
    Shuttle S01 | Status: Standby | Next Departure: 08:00 AM
.
.
.
.

.
[07:31:15] Server registry summary:
- B101: Connected | Active | On Route R1
- T22: Connected | Active | Delayed 20s ago
- U991: Connected | Private | Cannot be overridden
- S01: Connected | Standby | Not activated ye